The paper "Parallel Sorting on Symult 2010" has been presented and will appear in the following conference proceedings:

Proceedings of The Fifth Distributed Memory Computing Conference,
Charleston, South Carolina, April 9-12, 1990.

These proceedings will be available by July (1990) or so, they can be ordered from:

Jane Squires
DMCC5 Proceedings
Department of Mathematics
University of South Carolina
Columbia, SC 29208
(803) 777-7660

Yu-Wen Tung

Parallel Sorting on Symult 2010

P. Peggy Li
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109

Yu-Wen Tung*
USC – Information Science Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6695

Abstract

In this paper, three sorting algorithms, Bitonic sort, Shell sort and parallel Quicksort are studied. We analyze the performance of these algorithms and compare them with the empirical results obtained from the implementations on the Symult Series 2010, a distributed-memory, message-passing MIMD machine. Each sorting algorithm is a combination of a parallel sort component and a sequential sort component. These algorithms are designed for sorting M elements of random integers on a N-processor machine, where $M > N$. We found that Bitonic sort is the best parallel sorting algorithm for small problem size, $(M/N) < 64$, and the parallel Quicksort is the best for large problem size. The new Parallel Quicksort algorithm with a simple key selection method achieves a decent speed-up comparing with other versions of parallel Quicksort on similar parallel machines. Although Shell sort has a worse theoretical time complexity, it does achieve linear speedup for large problem size by using a synchronization step to detect early termination of the sorting steps.

Introduction

As indicated by Knuth in his famous book on sorting and searching [1]:

> *It would be nice if only one or two of the sorting methods would dominate all of the others, regardless of the application or the computer being used. But in fact, each method has its own peculiar virtues. ....*

This remains true, if not more so, for sorting algorithms on parallel machines for two reasons. First, the performance of a parallel sorting algorithm depends on the degree of parallelism it can exploit on a given architecture. For example, it often makes difference on level of parallelism one could exploit on an SIMD and on an MIMD machine. Second, the performance also depends on the speed of certain critical operations the underlying parallel machine could deliver. For example, interprocessor communication could be a dominating operation for distributed-memory machines because parallel sorting algorithms often require the same order of magnitude of communication steps as that of computation.

In this paper, we focus on only a class of MIMD machine on which the issue of interconnection network is not very important, and the communication speed is nearly balanced with the computation speed. By choosing such seemingly general-purposed, yet real, machine, we are able to concentrate on finding which sorting methods, or combinations of sorting methods, are possibly among the fastest on an MIMD machine.

We shall also confine ourselves to sorting a long list of random input data using less number of processing elements (or nodes). That is, the sorting problem we are interested in is to sort M elements of random integers on an N-node MIMD machine, where $M > N$. Initially, M unsorted elements are evenly distributed to each computation node. Each node operates on its own set of data independently, but can send or receive data from another node. When all nodes terminate, each node should hold a chunk of sorted list, and chunks are stored in consecutive order across all nodes such that the smallest chunk is stored in the first node and so on. Chunk size may or may not be M/N depending on the algorithm used.

Because of the problem nature $M > N$, each of the three sorting algorithms we have implemented is a combination of parallel sort (across nodes) and sequential sort (for local list). We used a parallel version of Quicksort [2], Batcher's bitonic sort [4], and a mixture of Shell's sort and odd-even transposition sort [1] as our parallel sorting strategies, and the UNIX/BSD qsort routine as the sequential sorting method. For simplicity, we shall call our algo-

*Supported by NASA Cooperative Agreement NCC-2-539 and RADC contract F30602-88-C-0135

rithms Bitonic sort, Shell sort and parallel Quicksort, respectively, in the following text.

Quicksort is not only a fast sequential sort method, it is also a parallel method by its divide-and-conquer nature. The only potential problem with the efficiency of a parallel Quicksort is the selection of its splitting keys. If such keys are randomly selected, the input list can be divided into uneven sublists and cause load unbalancing. Carefully calculated splitting keys will solve this problem but the extra calculation becomes a cost itself. So an efficient implementation needs to strike a balance between two extremes, which is, fortunately, not very hard to achieve. Impressive results for parallel Quicksort have been reported for a vector machine CDC STAR [3] and hypercube-interconnected MIMD machines [5], among others.

Batcher's bitonic sort [4], on the other hand, has been widely used across almost all kinds of parallel computers – sorting networks, hypercube machines [6], two-dimension mesh machines [9], SIMD machines [7] for its simplicity and stability. It has a time complexity $O(\log^2 M)$ for sorting M elements, which is reasonably efficient. The less known Shell sort is also selected because it appears to be a very efficient algorithm when implemented on Caltech/JPL's Hypercube machine [5].

In the rest of the paper, we will first introduce the underlying machine we used in our study, and its computation and performance model; followed by the three sorting algorithms and their time complexities. Then, we will discuss our empirical performance result, and address a few related issues such as how general our result can be, and what other sorting methods may also be considered.

Computation and Performance Model

The Symult Series 2010 system (S2010) is a distributed-memory message-passing MIMD computer consists of up to 1024 computational nodes interconnected by a high speed message-routing network (GigaLink). Each computational node has a Motorola MC68020 microprocessor as its CPU, operating at 25 MHz and augmented by the Motorola 68881 floating-point co-processor. A SUN-3 workstation is used as the front-end computer. The operating system on the S2010 nodes is called Reactive Kernel, and the programming environment on the front-end computer, serving as the interface between the users and the S2010, is called Cosmic Environment [8].

S2010 is facilitied with a fast communication network, called GigaLink network. A custom-designed message routing chip — Automatic Message Routing Device (AMRD) — provides fast fixed-route point-to-point message routing using "worm-hole" routing algorithm. The interprocessor communication rate is 13MB/sec regardless of the distance between source and destination. This feature makes the S2010 resemble to a fully-connected machine.

To characterize the machine behavior, we carefully measured timing for many computation and communication instructions. Here are some of the timing results useful for our sorting analysis, where one integer is equivalent to four bytes:

- copy one integer from one memory location to another, without taking memory allocation overhead into account, takes about 0.45 μs;
- memory allocation overhead per memory copy function (*bcopy*) is about 8 μs;
- comparison-exchange for two integers takes about 6.8 μs;
- transmitting one integer in a typed message from one node to another, without taking overhead into account, takes about 0.31 μs under low to normal traffic load.
- average overhead for sending a typed message from one node to another takes about 251 μs.

In other words, if routing a message with size K (integers) takes time $T_{route}(K)$, copying a same size message locally takes time $T_{copy}(K)$, and performing compare-exchange on K integers takes time $T_{comp-ex}(K)$, then

$$\begin{aligned} T_{route}(K) &= T_{route-overhead} + T_{route-int} \cdot K \\ &= 251\mu s + 0.31\mu s \cdot K \\ T_{copy}(K) &= T_{copy-overhead} + T_{copy-int} \cdot K \\ &= 8\mu s + 0.45\mu s \cdot K \\ T_{comp-ex}(K) &= 6.8\mu s \cdot K \end{aligned} \quad (1)$$

So we could conclude that, on S2010, the overhead for each message send/receive is very large, but the transmission speed is comparable to that of memory access speed. Therefore, in our sorting algorithms, the interprocess communication is a dominating term when M/N is small, and it graduately reduces its effect when M/N gets larger. Assuming the number of compare-exchange steps is the same as the number of messaging, then the communication overhead (i.e. $T_{route}(K)/T_{comp-ex}(K)$) is 41.4% for $K = 100$ and drops to 8.2% for $K = 1000$.

In the following timing analysis, we shall assume that each element to be sorted is represented as an integer, for simplicity. Thus K means the number of elements in each step of computation.

The sequential *qsort* routine takes an important role in all three algorithms, it has a time complexity $O(K \log K)$ for a single S2010 node to sort K elements. By experiments, we found that the timing equation for *qsort* with random input data can be represented as follows:

$$T_{qsort}(K) = O(K \log K) = 8.5\mu s \cdot K \log K \quad (2)$$

Bitonic Sort

In our implementation of this algorithm, the machine is configured as a N-node hypercube. Initially each node has M/N unsorted elements. Each node first sorts its data internally using the qsort routine, and then performs $(\log N \cdot (\log N + 1))/2$ steps of compare-exchange operation along all dimensions of the cube. After running the algorithm, every nodes have M/N elements sorted both locally and globally.

Our algorithm for each individual node is shown below, where *dim*, *my_nid*, and *mask* are the dimension of the cube, the node id, and a mask flag for selection of nodes, respectively:

1. Sort the (M/N) elements locally in each node using a *qsort*. Sort in ascending order if *my_nid* is even, in descending order if *my_nid* is odd.

For $i := 0$ to $(dim - 1)$ step 1 do (2), (3), and (4)

2. If the (i+1)-bit of my binary address is 1, $mask := 1$; otherwise, $mask := 0$.
3. For $j := i$ to 0 step -1 do
   (a). exchange my (M/N) elements with my j-th bit neighbor; (b). compare/exchange the two lists and copy the smaller half into the data area if $mask =$ the j-th bit of my binary address; copy the larger half into the data area otherwise.
4. Locate the maximum (or minimum) of the bitonic sequence in each node and perform a merge on sublists of length M/N. The sorted sublist is in ascending order if $mask = 0$; otherwise, it is in descending order.

Since each node has M/N elements, the time complexity of step (1) is $O(\frac{M}{N}(\log \frac{M}{N}))$. Each compare-exchange iteration in (3) takes time $2T_{route}(\frac{M}{N}) + T_{comp-ex}(\frac{M}{N}) + T_{copy}(\frac{M}{N})$, and there are $(\log N(\log N + 1))/2$ iterations in total. Each merge operation in step (4) takes time $T_{comp-ex}(\frac{M}{N} + 2\log \frac{M}{N})$, and this operation is performed $\log N$ times. As a result, the total time for Bitonic sort, based on our timing equations (1) and (2), can be expressed with unit time μs as follows:

$$\begin{aligned} T_{Bitonic} &= O(\frac{M}{N}(\log \frac{M}{N})) + \\ &\quad (\log N(\log N + 1)/2) \cdot (510 + 7.87(\frac{M}{N})) + \\ &\quad 6.8 \log N(\frac{M}{N} + 2\log \frac{M}{N}) \\ &= 8.5\frac{M}{N}(\log \frac{M}{N}) + \\ &\quad (241.4 + 3.94\frac{M}{N})\log^2 N + \\ &\quad (13.6\log M + 10.74\frac{M}{N} + 255)\log N \end{aligned} \quad (3)$$

The empirical timing curve for $N = 16$ is shown in figure 1.

Shell Sort

As described earlier, here Shell sort means a method that combines Shell's method and odd-even transposition sort as internode sort, and qsort as sequential sort. This algorithm, as well as the parallel Quicksort algorithm, are to be executed on a ring topology – the sorted data will be stored in the same way as in the Bitonic sort case, but with a slight difference that node address is arranged in ring topology. Both hypercube and ring topologies can be easily configured on the S2010, without significant performance difference.

This algorithm has three steps:

1. Sort (M/N) elements locally in each node with qsort.
2. Do a compare-exchange operation between pairs of adjacent nodes along the i-th cube dimension, for $i = \log N - 1, \ldots, 0$.
3. Do compare-exchange operations between pairs of adjacent nodes in the ring topology until no exchange is made in all the node.

The first part is a Shell's sort except that only one compare-exchange operation is performed in each hypercube dimension and the result list is partially sorted. This part takes $\log N$ compare-exchange operations in total. The second part is an odd-even transposition sort which terminates when no data is exchanged in all the node.

The number of odd-even transposition steps is equal to the maximal distance of a mispositioned element to its sorted position. After the diminishing-increment steps, the worst-case maximal distance is $(N - 2\sqrt{N} + 1)$, where N is the number of nodes.

Given an arbitrary element a, assuming that y and x are the addresses of the nodes that a is located after step (1) and after sorting, respectively. Thus, $|y-x|$ is the number of odd-even transposition steps required to move a to its final position. Let a be such an element that has maximal $|y-x|$ and $x < y$, now we like to find the minimal x for a given y. Let the binary address of y be $(y_1, y_2, \ldots, y_d)$, where $d = \log N$, i.e., the dimension of the cube. After step (1), all the elements in the nodes whose addresses can be derived from y by changing one or more y_i's from 1 to 0 should be smaller than the elements in y. If there are k "1" bits in the binary address of y, there will be at least $(2^k - 1)$ nodes in which the elements are smaller than those in y after step (1). Thus, the minimal element that may be located in node y after step (1) is always greater than the elements in the first $(2^k - 1)$ nodes after sorting. In other words, the minimal a in node y will be stored in the 2^k-th node $(x = 2^k - 1)$ after sorting is done. To maximize $|y - x|$, we shall find the maximal y with a proper k. Obviously, the maximal y which has k "1" bits is the one having all 1's in the most significant bits and $y = N - 2^{(d-k)}$. So $(y - x) = (N - 2^{(d-k)} - 2^k + 1)$ and the maximal $(y-x)$ is equal to $(N - 2^{d/2+1} + 1)$ or $(N - 2\sqrt{N} + 1)$ when $k = d/2$.

Therefore, in the worst case, there are $(N - 2\sqrt{N} + 1)$ compare-exchange operations in step (3). Each compare-exchange operation in step (2) and (3) takes the same time as one iteration in Step (3) of Bitonic Sort, i.e. $2T_{route}(M/N) + T_{comp-ex}(M/N) + T_{copy}(M/N)$. Therefore, the worst case time complexity of the Shell sort is

$$T_{Shell} = 8.5\frac{M}{N}\log\frac{M}{N} + (N - 2\sqrt{N} + \log N)(510 + 7.87\frac{M}{N}) \qquad (4)$$

The first term is the time for sequential sort of the local M/N element sublist. The real timing for the case $N = 16$ is shown in figure 1.

Parallel Quicksort

The quicksort is a divide-and-conquer sorting algorithm which is potentially applicable to parallel computation. In order to get the best performance of the quicksort, the splitting keys should be selected with great care so that the list to be sorted can be decomposed into two sublists of equal length. This fact is even more important in the parallel quicksort because the improper selection of the splitting keys results in load imbalance and the computation time is determined by the slowest node.

Similar to Bitonic sort and Shell sort, the unsorted list is stored evenly across the cube initially, i.e., each node has M/N elements in arbitrary order. After sorting, the sorted list will be stored in the cube in consecutive order but each node may have different number of elements. The parallel quicksort works as follows: First, $(N-1)$ splitting keys are selected using a presorting algorithm. Second, the list in each node is split into two parts according to a proper splitting key and exchanged with its neighbor along a certain dimension. This splitting process repeats $\log N$ times. At last, each node sorts its local list with a fast sequential sorting algorithm.

The algorithm and its time-performance are described as follows:

1. Choose k samples randomly from the $\frac{M}{N}$-element sublist of each node. Find the largest and the smallest elements in the sample, let them be (max, min).
2. Perform the maximum and minimum operations on each node's (max, min) pair globally across the nodes to find the maximum and the minimum elements, say $(gmax, gmin)$, in the whole sample. This global operation is done in a binary tree manner, and it needs $\log N + 1$ communication steps, i.e., $(\log N + 1) \cdot (T_{route}(2) + T_{comp-ex}(2))$.
3. Equally divide $(gmax - gmin)$ into $N - 1$ intervals and use the N boundary elements as the splitting keys. Since each node only needs $\log N$ splitting keys, each node can use a binary search to find all its keys in $\log N$ steps.
4. for $i := dim - 1$ to 0 step -1 do
   compare my sublist with the i-th splitting key and divide it into two sublists.
   if $(my\_nid < neighbor[i])$
   exchange the larger sublist with the smaller sublist of my i-th bit neighbor
   else
   exchange the smaller sublist with the larger sublist of my i-th bit neighbor
   endif.
5. Sequential sort the sublist locally.

The main part of the parallel Quick sort, i.e, step (4), takes

$$\log N \cdot (2T_{route}(\frac{M}{N}) + T_{comp-ex}(\frac{M}{N}))$$

for the best case, assuming each node always hold M/N elements after each compare-splitting step.

With a good set of splitting keys, the parallel Quicksort has a best/average time performance:

$$T_{Quick} = 8.5\frac{M}{N}\log\frac{M}{N} + 7.41\frac{M}{N}\log N + 767.2\log N \qquad (5)$$

The sampling time in step 1 is proportional to the sample size in each node, which is negligible. The first term of the equation is the sequential sorting time in step (5). And the real timing in the case of $N = 16$ is shown in figure 1.

Performance Comparison and Analysis

We have measured execution time for each of the above three sorting algorithms for $N = 8$, 16, 32 and 64, and M ranges from 2^6 to 2^{19}. Figure 1 shows the timing curves with the execution time versus $\log M$ for $N = 16$.

Figures 2 and 3 are speedup curves calculated from real execution time of the three parallel algorithms for $N = 16$ and 64, respectively.

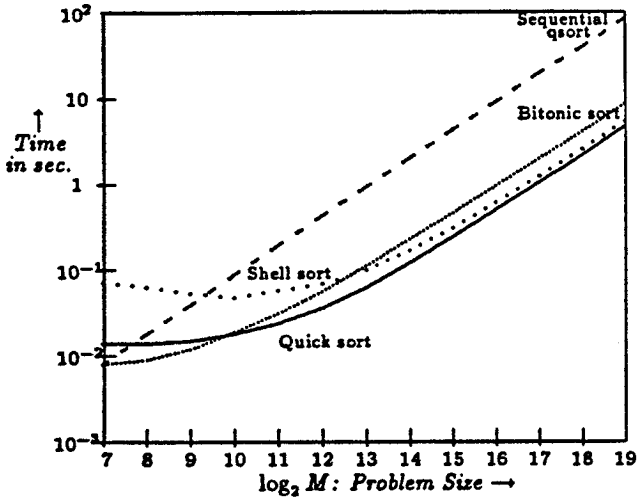

Figure 1. Timing on a 16-node S2010

Input lists are generated by using UNIX *random* routine. The execution time is determined by the sorting time of the slowest node. The down-loading and up-loading (input and output) time is not considered in our experiments.

From these speed-up curves, it is observed that the increasing communication overhead degrades the sorting speed of small lists (for lists with 1K elements or less) when the machine size increases. In the case of Bitonic sort, which has the lowest communication overhead and is the fastest algorithm for small lists,

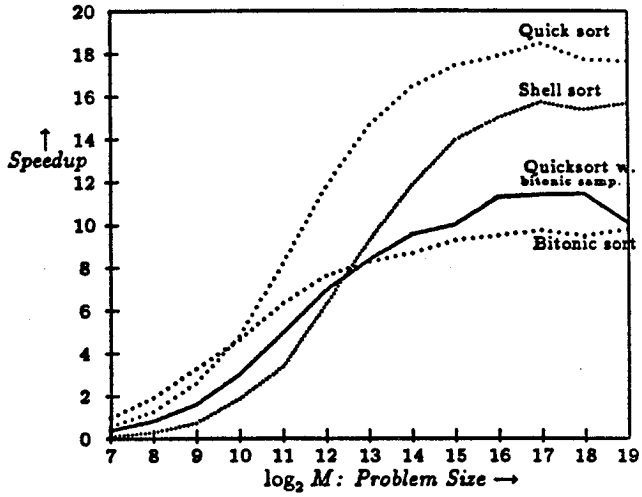

Figure 2. Speedup curves on a 16-node S2010

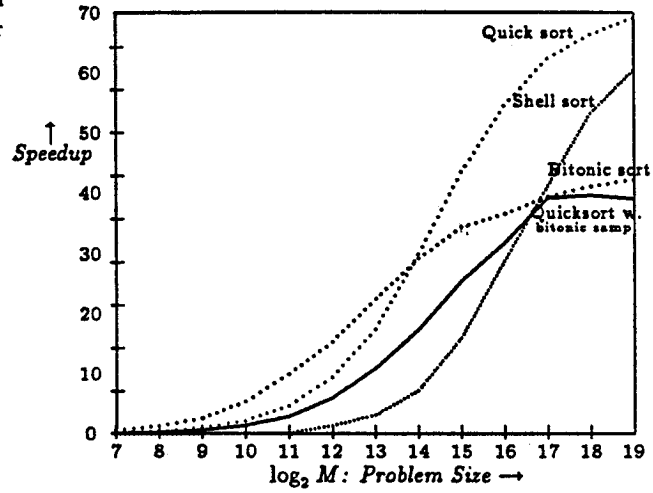

Figure 3. Speedup curves on a 64-node S2010

the ratio of the communication time to the computation time is more than two for small lists.

In the case of Shell sort, the timing equation 4 does not include the time to broadcast the boolean flag which indicates if any exchange has been made in each compare-exchange step in (3) of the Shell sort algorithm. This value may be negligible when $M \gg N$, but becomes the major overhead when N gets large, or when $M \simeq N$. Broadcasting is done in binary tree manner which requires $(\log N + 1)$ steps of message transmission after each compare-exchange step. For the worst case, the broadcast overhead is as high as $253.3(N - 2\sqrt{N} + 1)(\log N + 1)$. Although the parallel sorting part of T_{Shell} has an $O(M)$ time complexity in the worst case, the broadcasting step may save a lot of

compare-exchange steps for random data input. The empirical result shows that the parallel Shell sort can achieve linear speed-up for large problem size random data. See figures 2 and 3.

As to parallel Quicksort, empirical result shows that a presorting procedure as simple as the abovedmentioned splitting key selection mechanism can result in very good load balancing, thus a super linear speedup is observed. A more complicated presorting algorithm based on the bitonic sort has also been attempted, but it results in a higher overhead, i.e., $O(k \log^2 N)$, for k samples each node, and a worse load balancing than the above algorithm. Consequently, we can conclude that for random data, the simple equally-divided key selection method can achieve the best performance of the parallel Quicksort. See figures 2 and 3.

Unlike the other two sorting algorithms, the sorted list obtained from this algorithm is not evenly distributed in each node. This is not a problem if the sorted list is up-loaded to the host machine without further computation. On the other hand, if sorting is just a part of the computation and the sorted list needs to stay in the cube for later use, the unbalanced data distribution may not be desirable. In this case, we may need to rearrange the elements so that each node keeps the same number of elements. The cost for the redistribution needs further investigation.

Conclusions

We have implemented three sorting algorithms on S2010, a distributed-memory message-passing MIMD machine. These algorithms are chosen because they can be parallelized easily on a mesh or hypercube architecture. Each sorting algorithm is a combination of parallel and sequential sorting methods and has a different time complexity.

In the parallel sorting component, Bitonic sort takes a fixed number of steps to sort despite of the input data pattern, with time complexity $O(\frac{M}{N} \log^2 N)$. Parallel Quicksort has a performance that depends on how good splitting keys are selected, and it is shown that with a little overhead of presorting this algorithm can achieve very good load balancing, and thus a best time performance $O(\frac{M}{N} \log N)$. The performance of the Shell sort is constrained by its second part, the odd-even transposition sort, which is a slow sequential sorting algorithm. Nevertheless, by taking the advantage of the asynchronous nature of S2010, the parallel version of the Shell/odd-even transposition sort can be as good as the parallel Quicksort when $M \gg N$.

In the sequential sorting part, which is performed on the M/N elements locally on each processor as the first step in Bitonic sort and Shell sort, or on varied number of elements locally as the last step in parallel Quicksort, has a time complexity $O(\frac{M}{N} \log \frac{M}{N})$.

The overall performance of the three algorithms is a combination of this sequential performance and the parallel sort performance. We found from our empirical results, for relatively small size of problems, $M/N < 64$ say, Bitonic sort is the best because it has the lowest synchronization overhead in the algorithm. The parallel Quicksort is the best for large problem size, which agrees with our analysis. It is interesting to learn that Shell sort outperforms Bitonic sort in the case of large problem size, which is mainly due to the fact that the Shell sort often terminates the sorting steps earlier. Both the parallel Quicksort and Shell sort achieve linear speed-up comparing to sequential qsort for large problem size on 8 to 64 processor machines. Shell sort is the slowest among all for the small problem sizes because of its high synchronization overhead.

References

[1] Knuth, D.E., *The Art of Computer Programming, Vol.3, Sorting and Searching.* Reading, MA: Addison-Wesley, 1973.

[2] Hoare, C.A.R., "Quicksort," Computer J., vol. 5, no. 1, 1962, pp.10-15.

[3] Stone, H.S., "Sorting on STAR," IEEE Trans. on Software Engineering, Vol.SE-4, No.2, March 1978.

[4] Batcher, K.E., "Sorting Networks and Their Applications," in Proc. AFIPS 1968 SJCC, vol. 32, Montvale, NJ, AFIPS Press, pp.307-314.

[5] Felten, E., Karlin, S. and Otto. S., "Sorting on a Hypercube," unpublished Caltech report Hm-244, 1986.

[6] Johnsson, S.L., "Combining Parallel and Sequential Sorting on a Boolean N-Cube," Proceedings of the 1984 International Conference on Parallel Processing, 1984, pp.444-448.

[7] Tung, Y.-W. and Mizell, D.W., "Two Versions of Bitonic Sorting Algorithms on the Connection Machine," Third Annual Parallel Processing Symposium, Fullerton, California, March 1989.

[8] "Programmer's Guide to the Series 2010^{TM} System – Preliminary," symult Systems Corp., January 1989.

[9] Nassimi, D., and Sahni, S. "Parallel Permutation and sorting algorithms and a new generalized connection network," JACM 29(3), 642-667, 1982.